La primera revista de PC para los desarrolladores de videojuegos



Sumario

Universal Music en la Web

Por fin podemos disfrutar en la web de la discográfica que trae hasta nosotros la música de artistas de la talla de Raimundo Amador, Lucrecia, Blues Brothers 2000, Molotov o Aqua. Dentro de las páginas de esta compañía aparecen apartados como artistas, lanzamientos, noticias, giras, etc. Mención especial merece el canal de comunicación interactiva, con secciones como Top 5, una lista de éxitos confecciona con los votos de los visitantes, o JukeBox, en la que se pueden escuchar *on-line* los éxitos de la compañía. Icon Medialab ha sido la encargada de asesorar a Universal Music en este terreno.

Llega a Barcelona la feria multimedia electrónica más importante de Europa





Las empresas MDI y Dream Comunicación han llegado a un acuerdo para celebrar en Barcelona (España) la feria del grafismo, producción y multimedia electrónica más importante de Europa, el CGIX. La importancia de este evento es comparable a la celebración de la feria Siggraph en los Estados Unidos.

Del 2 al 6 de febrero de 1999, la Feria de Barcelona, situada muy cerca del aeropuerto, acogerá a las compañías más importantes en animación, diseño gráfico, *broadcast*, multimedia e Internet para que muestren sus nuevos productos e innovaciones. Para coordinar la CGIX de Barcelona, la empresa Dream Comunicación ha sido escogida Agente Oficial para España por la compañía MDI, organizadora del evento. La función de Dream Comunicación será la organización del Pabellón Español que reunirá a las empresas más destacadas del sector. En este sentido, Dream Comunicación se encargará de la coordinación del alquiler del espacio y de los *stands*. En sus manos también estará la organización de las diferentes conferencias y la realización de la cobertura del evento a través de una web y de un diario bilingües (inglés/español).

Para realizar la cobertura informativa del evento, Dream Comunicación desplazará su redacción al centro del CGIX. Periodistas, fotógrafos, cámaras de vídeo... La finalidad será realizar "in situ" una web y un diario que recoja las presentaciones más importantes, las entrevistas más interesantes y los datos más relevantes de este acontecimiento.

La última edición de la CGIX se celebró en Amsterdam (Holanda) en enero de este año. Según una encuesta realizada entre los más de cinco mil visitantes profesionales que asistieron, un 80 por ciento declaró que tenían intención de compra a realizar entre 6 y 12 meses. A medida que se vayan acercando las fechas de celebración del CGIX de Barcelona, se irán conociendo los periodos de inscripción y precios, así como todas las novedades que se produzcan.

Madge Networks anuncia un paquete de productos Token Ring

Este suministrador internacional de soluciones de redes presenta un paquete de producto Token Ring de alta velocidad de 100 Mbps, con todas las funcionalidades basados en estándares. La nueva serie de productos, que permite desarrollar HSTR en todas las redes, incluye el adaptador Madge Smart 100/16/4 PCI-HS Ringnode. Se trata de una tarjeta de interfaz de redes que permite la ampliación a 4, 16 o 100 Mbps; soporta, además, la detección y selección automática de velocidades y tipos de cables, de forma que se adapta a cualquier entorno sin necesidad de reconfigurar el equipo. También ofrecen módulos HSTR de fibra de dos puertos y de cobre de cuatro puertos para la familia Smart Ringswitch de conmutadores Token Ring. Por otra parte, los precios a los que aparecerá la serie al mercado serán agresivos, de modo que serán bastante asequibles. Madge se convierte de esta manera en el primer fabricante que nos presenta aplicaciones de redes troncales Token Ring de 100 Mbps. Las primeras unidades que se encuentren disponibles serán para el conmutador Smart Ringswitch Express y para el conmutador Smart Ringswitch Plus de alta capacidad, en sus versiones de cobre o fibra.





En nuestro CD-Rom de portada incluimos el siguiente material:

- **Los códigos fuentes de los ejemplos comentados en el artículo 3D Manía.**
- **GIF Movie Gear 2.5: Una herramienta que sirve para manejar imágenes.**
- **Anarkano: Un excelente ejemplo de videojuego de programación sencilla.**

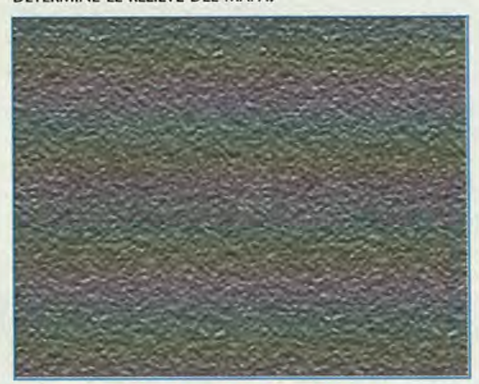
Autor: Jesús de Santos
icg_ent@bigfoot.com

Bump Mapping

Este mes comenzamos una serie de artículos donde descubriremos la técnica Bump-Mapping, muy utilizada en escenas 3D para simular relieve en los polígonos.

Uno de los principales problemas con los que se encuentran tanto diseñadores como grafistas es que el número de polígonos que se pueden emplear para una escena está siempre muy limitado por el hardware. Hemos de recordar que estamos diseñando mallas que van a ser empleadas en programas que trabajan en tiempo real. Por tanto no podemos emplear cualquier número de polígonos como haríamos en una producción cinematografica. Generalmente el grafista siempre requiere un número mayor de polígonos que el permitido, mientras que el programador siempre intenta reducir la complejidad poligonal a toda costa para conseguir acelerar al máximo el engine. Esta es la eterna disputa entre programadores y grafistas. Esta situación ocurre especialmente en programas que deben «renderizar» en tiempo real, como por ejemplo la programación de videojuegos, tema al que nos referimos habitualmente en esta sección. La cosa se complica aun más si nos situamos en un ordenador personal. Hay que reconocer que con las tarjetas aceleradoras 3D la calidad ha subido mucho en el último año pero, aun así, nos encontramos ante una falta de recursos y de máquina para poder mover la escena de nuestros sueños en tiempo real.

FIGURA 1. ESTA ES LA TEXTURA A LA QUE QUEREMOS DAR RELIEVE. VAMOS A RELACIONAR EL COLOR DE CADA PÍXEL CON SU ALTURA RELATIVA. ESTA ALTURA SERA LA QUE DETERMINE EL RELIEVE DEL MAPA.



Generalmente gran parte de los polígonos tienen que ser empleados en dar la forma principal al objeto mientras que el modelado de los detalles se descuida a menudo recurriendo al empleo de texturas que suelen enmascarar bastante bien esta deficiencia. Así, por ejemplo, el diseño poligonal del relieve de una pared de mármol es absolutamente impensable. La solución habitual es emplear una textura de mármol que generalmente cumple bien su misión. Del mismo modo, los ladrillos de una pared suelen realizarse a base de textura; es impensable modelar por polígonos estos aspectos. Pero si somos un poco más ambiciosos podremos conseguir el deseado detalle sin gastar un número adicional de polígonos. La solución se llama Bump-Mapping. Cuando nos referimos a detalle, queremos decir que los algoritmos de iluminación tendrán un comportamiento similar al que produciría una malla de polígonos equivalente. Bump-Mapping no es más que «truco» que mediante el empleo de varias texturas simula variaciones de normales dentro de un mismo polígono. Pero vayamos por partes.

El hardware limita en gran medida el número de polígonos por pantalla. Por ello se simulan los relieves

Hemos preferido tratar esta técnica con detalle y tranquilidad para que el lector entienda perfectamente el funcionamiento. Vamos a dedicar más de un artículo para mostrar el funcionamiento, así como diversas utilidades de esta habitual técnica.
El objetivo de estos artículos es claro: dotar a nuestro motor de la capacidad de mostrar texturas con Relieve o *Bump* (en polígonos tridimensionales). Eso es lo que conseguiremos al acabar la serie de artículos sobre este tema. En este primer artículo pondremos límites al problema y vamos a buscar solución para Bump-Mapping en 2 dimensiones. Una correcta compresion de este artículo es la clave para comprender el Bump 3D, que no es más que una mera extensión (caso similar a cuando explicamos el Clipping, primero en polígonos 2D y luego en 3D).

BUMP-MAPPING 2D

En este apartado nos vamos a limitar únicamente a las dos dimensiones. Así, nos olvidamos de los polígonos, de las cámaras, matrices, etc. Tendremos una textura (un fichero TGA, por ejemplo) en pantalla a la que cambiaremos de relieve. Vamos a trabajar por ejemplo con el dibujo que aparece en la Figura 1.
Queremos conseguir que los bultos que aparecen en el dibujo creen una sensación de relieve. Del mismo modo, las letras deben poseer un relieve propio para que resalten y se puedan leer.
Para almacenar estos datos empleamos lo que se denomina un mapa de alturas. Un mapa de alturas es similar a un *bitmap*, pero en vez de almacenar el color en cada posición almacenamos la altura de cada posición. Así, si por ejemplo empleamos alturas de 8 bits (lo cual suele ser suficiente), un valor de *0* indica mínima altura mientras que el valor *255* indica la máxima altura. Por supuesto todos estos convenios son arbitrarios. Lo importante es que seamos coherentes a lo largo de todo el proceso con las decisiones que tomemos inicialmente. Éstos son los valores que vamos a emplear para crear el relieve. Una zona donde todos las posiciones tengan la misma altura no deberá mostrar relieve mientras que una zona con muchas variaciones deberá dar sensación de rugosidad. Del mismo modo deberemos tener en cuenta la variación de nivel de una posición con respecto a sus vecinos. Vamos a considerar los vecinos de una posición a las casillas que tenemos a los lados derecho, izquierdo, superior e inferior. No tenemos en cuenta los vecinos de las diagonales.
¿Cómo construir el mapa de alturas? No suele ser muy difícil a partir de la imagen a la que queremos aplicar el relieve. Normalmente, tomando el color como la altura suele funcionar (en imágenes con paleta si nuestro altura es de 8 bits). Así, nuestro mapa de alturas quedaría de un modo similar a la figura 1, si interpretamos que el color negro es la altura *0* y el color blanco la altura máxima.

A veces puede ser necesario que el mapa de alturas y el dibujo que queremos iluminar sean distintos. En ese caso emplearemos dos texturas distintas. Veremos este caso cuando tratemos el Bump Mapping en 3D.
El siguiente paso es la construcción de lo que se denomina el mapa de luz. El «mapa de luz» es otro *bitmap* que representa el foco de luz con el que vamos a iluminar nuestro relieve. Para ello empleamos una textura True-Color (por ejemplo) con cualquier forma y cualquier color. Lo que realmente se va a pintar en pantalla son pixels del mapa de luz, por lo que su color y forma va a determinar la imagen final del efecto. Para nuestro ejemplo, vamos a emplear una luz circular blanca como la que aparece en la figura 2.
Podemos considerar este mapa de luz como el equivalente a las texturas de entorno que empleamos en el último artículo para simular Phong. De hecho el *enviroment-mapping* jugará un papel importante en el Bump 3D.
Con esto ya tenemos todo lo necesario para crear nuestro efecto. Supongamos ahora que empleamos un mapa de alturas cuyas alturas son todas *0*. La imagen que deberíamos obtener sería la de la figura 2, es decir, el mapa de luz original, debido a que no existe ningún desnivel que altere el mapa de luz. Según van apareciendo relieves en forma de bultos el dibujo de la figura 2 debe perturbarse ligeramente para crear esa sensación de relieve. Precisamente en eso consiste la ténica de Bump-Mapping. Tenemos que introducir perturbaciones en nuestro mapa de luz para crear el efecto de relieve. Entendemos por «pertubación» un desplazamiento en las coordenadas del mapa de luz: si por ejemplo tuvieramos que pintar el píxel del mapa de luz *<50,50>* una perturbación nos podría obligar a pintar el *<55,55>*, que tendría una iluminación distinta. La perturbación debe poder ocurrir tanto horizontal como verticalmente y en ambos sentidos.
¿Cómo calcular la perturbación? Una buena idea es considerar el mapa de alturas como una gráfica *XY* y calcular la tangente en el eje *X* y en el eje *Y* en cada punto del mapa de alturas. Podemos calcular la tangente mediante la derivada. Una buena aproximación de la derivada:

Dx = Vecino Izquierdo - Vecino Derecho
Dy = Vecino Superior - Vecino Inferior

La derivada nos da una medida de la pendiente del punto con respecto a su vecindad. Con esta pendiente podemos perturbar el mapa de luz correctamente y conseguir el efecto deseado. A veces es buena idea multiplicar *dx* y *dy* por algún factor para escalar la pendiente. Por tanto *dx* y

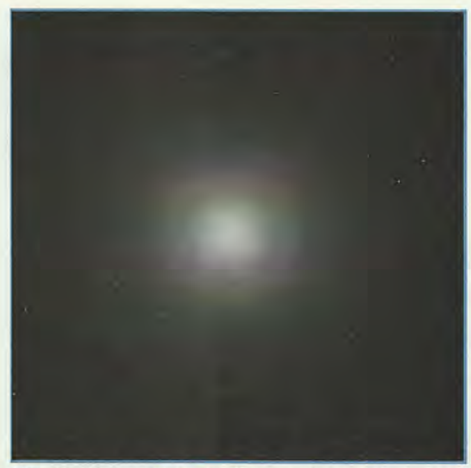


FIGURA 2. EL MAPA DE LUZ, MUY SIMILAR AL QUE EMPLEAMOS EN LA TÉCNICA DE ENVIROMENT-MAPPING PARA SIMULAR PHONG.

dy [números con signo] representan la pertubarción que aplicar en esa posición. *Dx* representa la perturbación horizontal mientras que *Dy* representa la perturbación vertical. Ambas perturbaciones con signo.
Pongamos el siguiente ejemplo: supongamos que en un punto de pantalla corresponde iluminar con la posición *<50,50>* del mapa de luz. Este punto posee unos valores de *dx=-5* y *dy=1*. Por tanto, el valor real que debemos copiar en pantalla es el *<50+dx,50+dy> = <45,51>*. Con esta sencilla operación podemos conseguir el efecto de relieve deseado. En la figura 3 tenemos el efecto final que conseguimos aplicando BumpMapping 2D.

Para comprender plenamente el BumpMapping 3D hay que empezar por analizar el 2D

Es increible la calidad gráfica que se consigue con un algoritmo tan sencillo. Recomiendo al lector que ejecute ahora mismo el ejemplo que viene en el CD-Rom que acompaña a la revista para que pueda apreciar la calidad del resultado obtenido. Pasamos ahora mismo a describir con detalle los pasos llevados a cabo para construir dicho ejemplo.

EL EJEMPLO DE ESTE MES

Como todos los meses, con cada artículo encontramos su correspondiente ejemplo. Este mes hemos desarrollado un ejemplo bastante completo y hemos intentado sacar el máximo partido a las rutinas de lectura de TGAs que desarrollamos hace 2 artículos.
Nuestro objetivo en el ejemplo de este artículo es mostrar un efecto Bump-Mapping que funcione en ventana y a resoluciones de 15-16-24-32 bpp. Con la

LISTADO 1

```
void draw_bump(SDWORD x, SDWORD y,
BYTE *light, BYTE *bump,
                BYTE *vid, BYTE bytespp) {

// Parámetros de la función:
// - x,y posición de la esquina superior
izquierda del mapa de luz
// - light : mapa de luz
// - bump  : mapa de alturas
// - vid        : puntero a pantalla
// - bytespp : Bytes por pixel : 15,16,24,32

   DWORD preX,preY,preXvid,preYvid;
   SDWORD i,j;
   SDWORD xlen,ylen;
   DWORD cvid,clight;
   SDWORD dx,dy;
   DWORD up,down,left,right;

// Clipping del mapa de luz

   preX=x<0?-x:0;
   preY=y<0?-y:0;

   if(x<0) preXvid=0;
   else preXvid=x;

   if(y<0) preYvid=0;
   else preYvid=y;

   if(x+LIGHT_XSIZE>XRES) xlen=XRES-
preXvid;
   else xlen=x+LIGHT_XSIZE-preXvid;

   if(y+LIGHT_YSIZE>YRES) ylen=YRES-
preYvid;
   else ylen=y+LIGHT_YSIZE-preYvid;

   cvid=preYvid*XRES + preXvid;
   clight=preY*LIGHT_XSIZE + preX;

   ...
```

flexibilidad que dimos a nuestro cargador de texturas, esto no va a ser ningún problema. Así pues, nuestro primer paso consiste en reservar memoria, cargar las texturas y convertirlas al modo de vídeo actual. Debemos tener en cuenta que el mapa de alturas es realmente una textura con paleta. Podemos ignorar la paleta, pues en este caso no nos interesa su contenido.

```
   light=(BYTE
*)malloc(headL.width*headL.height*bytespp);
   bump=(BYTE
*)malloc(headB.width*headB.height*sizeof(BYTE)
+256*4);
```




```
read_tga_picture("Light.Tga", light,
headL.width, headL.height, &gfx);
    read_tga_map("bump2d.tga", bump, bump +
XRES*YRES, headB.width,
headB.height);
```

Queríamos añadir al programa algo de interactividad, así que incluimos la opción de mover el foco de luz con las teclas de los cursores, lo que además servía para poder apreciar mejor el efecto de relieve. Este añadido traía un problema: la necesidad de realizar el recorte del mapa de luz. Pero esto no es ningún problema, como vimos en nuestro artículo sobre Clipping 2D. Nosotros hemos empleado un mapa de luz de 512x512 sobre una ventana de 320x240 (todos estos parámetros pueden ser variados fácilmente en el código). El código encargado de realizar el clipping 2D aparece en el listado 1.
Una vez tenemos los valores necesarios obtenidos del clipping 2D del mapa de luz recorremos los píxels de pantalla y en cada uno de ellos calculamos las perturbaciones horizontales y verticales como hemos explicado más arriba. El inconveniente que encontramos aquí viene dado por los posibles desbordamientos que se pueden producir, por ejemplo al acceder al vecino izquierdo de la posición *<0,0>* (acceso a memoria prohibida) o de cualquier posición que se encuentre en *<0, y>0 >* (cálculo de la pendiente incorrecto). Con el objetivo de no relentizar mucho el efecto, nos preocupamos únicamente de no acceder a posiciones de memoria prohibidas descuidando el cálculo de las perturbaciones en los bordes del mapa. Al aplicar la perturbarción obtenemos un problema similar, esta vez con el mapa de alturas. De nuevo nos preocupamos únicamente de no realizar accesos a memoria que puedan provocar una excepción de página. Todo ello aparece claramente en el listado 2, que muestra el «loop» principal para el modo de vídeo de 15/16 bpp.
Hat qye destacar el empleo de la máscara 262143 usado en el Acceso al mapa de luz. Con esto nos aseguramos que no estamos leyendo fuera de la región 512x512. El empleo de texturas cuadradas y con dimensiones que son potencias de dos suelen ser muy útiles en estos casos.

LISTADO 2

```
case 2:        // 15 , 16 bpp

  for(i=0;i<ylen;i++,cvid+=XRES,clight+=512)

              for(j=0;j<xlen;j++) {

              if(cvid+j==0) left=cvid+j;
              else left=cvid+j-1;

              if(cvid+j==XRES*YRES-1)
right=cvid+j;
              else right=cvid+j+1;

              if(cvid+j<XRES) up=cvid+j;
              else up=cvid+j-XRES;

              if(cvid+j>=XRES*YRES-XRES)
down=cvid+j;
              else down=cvid+j+XRES;

              dx=*(bump+left) - *(bump+right);
              dy=*(bump+up) - *(bump+down);

              *(wvid+cvid+j)=*(wlight+

((clight+j+dx+dy*LIGHT_XSIZE)&262143));

        }

 break;

// Aquí vienen los casos de 24 y 32 bpp
```

El propio usuario es el que debe jugar con los valores y observar los distintos resultados que puede obtener

Para los casos de 24 y 32 bpp el código es similar. Así, por ejemplo, para 24 bpp realizamos 3 escrituras a memoria de vídeo: una para la componente roja, otra para la verde y otra para la azul. Esto es más rápido que realizar una escritura de WORD (2 bytes) y otra de BYTE, pues generalmente la escritura de WORD estará desalineada.
Ejemplo para 24 bpp:

```
    dx=*(bump+left) - *(bump+right);
    dy=*(bump+up) - *(bump+down);

*(vid+(cvid+j)*3)=*(light+((clight+j+dx+dy*LIGH
T_XSIZE)&262143)*3);

*(vid+(cvid+j)*3+1)=*(light+((clight+j+dx+dy*LI
GHT_XSIZE)&262143)*3              + 1);

*(vid+(cvid+j)*3+2)=*(light+((clight+j+dx+dy*LI
GHT_XSIZE)&262143)*3              + 2);
```

Como se puede ver, el presente código es muy similar para cualquiera de los tres tipos de modo de vídeo con los que trabajamos. La



FIGURA 3. ESTE ES EL RESULTADO FINAL. ES REALMENTE SORPRENDENTE LA CALIDAD GRAFICA DEL EFECTO FINAL.

única diferencia, básicamente, consiste en el empleo de los punteros adecuados. La inicialización correcta de estos punteros se realiza en las siguientes líneas:

```
WORD *wvid=(WORD *)vid;
WORD *wlight=(WORD *)light;
DWORD *dvid=(DWORD *)vid;
DWORD *dlight=(DWORD *)light;
```

Todo el código aparecerá junto al ejecutable dentro del CD-Rom que acompaña a la revista. Seguimos empleando nuestas rutinas habituales que trabajan con la API Direct X y que se compilan bajo Visual C++. Es aconsejable que aquellos lectores que quieran ver todas las posibilidades reales de este código juegue con los diversos valores que tiene y observe los distintos resultados que se pueden obtener con él. Desde luego, la calidad gráfica que se puede conseguir con el mismo es innegable.
Con esto terminamos nuestra primera entrega acerca del Bump-Mapping centrado en las dos dimensiones. En los próximos artículos de esta serie veremos muchos más efectos que podemos realizar y su aplicación al Bump-Mapping 3D que es el objetivo final de esta serie de artículos.
Recordamos que la dirección de correo del autor sigue abierta a todos vosotros para cualquier tipo de sugerencia que se os ocurra hacer tras leer estas páginas. Se responderá a todos los mensajes lo antes posible.

Fe de errores

Hemos descubierto algunos «bugs» en las rutinas de Tga, que ya ofrecimos hace tiempo. Sobre todo había algunos fallos serios en la función que se encargaba de capturar las pantallas. Son problemas derivados del portado a C++ y sobre todo debidos a descuidos del autor de estas líneas. Este mes entregamos un nuevo código fuente <tga.cpp> revisado y con todos los fallos corregidos. Gracias a todos aquellos que nos habéis avisado de estos problemas.

Autor: Antonio Marchal

Resolución de tipos de juegos

Seguimos con la nueva etapa de DIV dentro de la revista Game Over. Este mes continuamos hablando de la problemática de los distintos tipos de juegos. Veremos los juegos del tipo Shoot'em'up, conocidos en España comúnmente como matamarcianos.

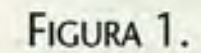
Todos hemos jugado alguna vez a algún Shoot'em'up. Pero este nombre proviene del idioma anglosajón. Su traducción podría ser algo así como «cárgatelos a disparos». En España, este tipo de juego es más conocido como matamarcianos.

FUNDAMENTOS

Los juegos de matamarcianos son de los más simples en su realización, aunque también se pueden complicar hasta límites insospechados. Es decir, si únicamente se quiere hacer un Shoot'em'up donde exista una nave protagonista que se dedique a eliminar hordas de enemigos, una detrás de otra, la programación de dicho juego puede ser muy simple, y más si los enemigos no son muy *inteligentes*. Pero si se quiere un juego más trabajado, se pueden incorporar multitud de detalles y mejoras. Algo fundamental en este tipo de juegos son los disparos. Tanto los disparos de la nave protagonista como los de los enemigos, si estos dispararan, pueden funcionar de distintas formas. Aunque lo más normal es que la nave protagonista dispare hacia arriba o hacia un lado, o en general hacia adelante, y los enemigos tengan bastante buena puntería al intentar acertar a la nave protagonista. Se pueden incorporar distintos tipos de disparo durante el juego, como misiles, disparo doble, mega-bombas, etc. Los distintos disparos pueden ser utilizables durante todo el juego, o únicamente ser accesibles mediante una serie de bonus, que son recolectables. Pero de los bonus hablaremos más adelante.

Algo fundamental en este tipo de juegos son los disparos

Una vez con el protagonista, que es la nave dirigida por el jugador, con los enemigos y los disparos, uno de los temas que el programador debe elegir es el modo de visión del videojuego. Normalmente las vistas más usadas son la aérea y la lateral. También puede que se quiera elegir entre si existe movimiento de pantalla o, como también se denomina, *scroll*. Si existe dicho movimiento, puede ser vertical, horizontal e incluso multidireccional.
Otro de los temas es si se dispondrá de decorado. Pero no solo hablamos de un decorado de fondo, sino que lo que se intenta comentar es si existiera un decorado con el que el jugador pudiera interactuar. Es decir, que se pueda chocar, o simplemente que le haga un efecto de pared. Además, como ya se comentó, existe la posibilidad, muy utilizada en estos juegos, de los bonus. Son una especie de objetos que el jugador puede recoger, y que le otorgan a la nave protagonista una serie de mejoras, como distintos tipos de disparos, inmunidad, o mayor capacidad de destrucción. También, en algunos juegos, existen una serie de enemigos, que son más difíciles de matar, y que normalmente están al final de la fase. Siempre y cuando el juego tenga distintas fases y niveles.


FIGURA 1.



PROBLEMATICA

En este apartado vemos los distintos problemas, de forma más detallada, que nos podemos encontrar al programar un Shoot'em'up. El primero de ellos es el movimiento del protagonista, cuya forma de funcionar debe elegir él mismo. Este movimiento puede ser en una única dirección o multidireccional. También puede darse el caso de que la nave protagonista se mueva según se pulse el teclado o girando. Es decir, en el primer caso, si se pulsa la tecla de arriba, la nave se moverá hacia arriba; y en el segundo caso avanzará hacia delante, y cuando se pulse las teclas de izquierda y derecha girará.
Aparte de este movimiento de la nave, hay que tener en cuenta el movimiento del juego en sí. Es decir, si tendrá o no *scroll* o movimiento de pantalla, y hacia qué dirección se realizará tal movimiento. Puede darse el caso de que, dependiendo de la fase, el movimiento del juego en general varíe, pero esto es elección del programador, como casi todo.
Siguiendo con los movimientos, seguiremos hablando esta vez de los enemigos. El tipo de movimiento se podría dividir en dos. Bien con trayectoria, bien inteligente, o semi-inteligente. Con el primero de ellos se puede diseñar un editor de trayectorias. El que denominamos inteligente es aquel que tiene una especie de seguimiento del protagonista o de cualquier otro elemento. También puede haber enemigos que no se muevan, pero estos ya lo veremos cuando hablemos del decorado.
Una vez vistos estos elementos comentaremos los disparos. El primero de ellos será el disparo simple que tiene el jugador, aunque también puede existir el disparo múltiple. Incluso se podrían incluir misiles, bombas y megabombas. Además de estos disparos, por supuesto estarán los de los enemigos. Estos normalmente irán destinados a la nave protagonista, es decir, apuntando a dicha nave.

Otro tema problemático es si se quiere incluir un escenario, además del que haga las veces de fondo. Es decir, una serie de paredes que evitan que la nave pase por ese punto. Puede que dicho muro sea o no peligroso para el protagonista. Es decir, se puede elegir entre que el escenario quite o no energía a la nave protagonista. Además, se pueden integrar enemigos no móviles, es decir, que estén integrados dentro del escenario. Por lo tanto, si se tiene escenario se debe controlar el choque o colisión con el mismo. Y se deberían detectar, lógicamente, las colisiones con los enemigos. Incluso se puede dar el caso de que parte del escenario o enemigo sea peligroso para el protagonista, y por lo tanto le quite energía, y otra parte del mismo sea indiferente, es decir que el protagonista pueda pasar por encima.

Cambiando de tema, pasemos a los bonus. Se pueden conseguir al pasar por ciertas partes del mapa de juego, o cuando se elimine a una serie de enemigos. Estos bonus, pueden dar distintos tipos de disparos, vidas, o incluso escudos. Y hablando de escudos, estos pueden tener multitud de formas, ya que pueden ser totales, parciales o incluso siendo estos los más complicados de programar, móviles, pudiendo por ejemplo dar vueltas alrededor de la nave.

Por último comentar los cambios de fase, si el juego dispusiera de ellas. Y si se diera el caso, también los enemigos de fin de fase. Estos suelen ser de gran tamaño, y más difíciles de matar que el resto. Tanto este enemigo como la nave protagonista pueden tener un nivel de energía. Normalmente, cuando una nave es alcanzada automáticamente es destruida, aunque se podría implementar una barra de energía que les diera una longevidad mayor.

Con esta enumeración quedan explicados todos los problemas que nos podemos encontrar al programar un juego de este tipo. A continuación, presentamos las soluciones DIV para estos problemas.

RESOLUCION

El primer problema que nos encontramos es mover la nave protagonista. Se puede optar por un movimiento direccional o relativo. Entendiendo como movimiento direccional aquel en el que cuando se pulsa la tecla derecha la nave se dirigirá a dicha dirección. En cuanto al movimiento llamado relativo lo podríamos denominar como aquel que se realiza del siguiente modo. Cuando se pulse las teclas de izquierda o derecha, el protagonista girará, y avanzará cuando se pulse la tecla de dirección arriba.



FIGURA 2.

Para poder mover con movimiento direccional, únicamente se deben alterar las coordenadas del proceso protagonista. Es decir, sus variables locales X e Y, incrementando o decrementando según convenga. Si se usa el otro método, usaremos la variable local *angle* para que el protagonista gire. Además, también se debe usar la función *advance()* para cuando queramos la nave avance. Con esto queda visto el movimiento de la nave, ya que no suelen utilizarse otros efectos, como la inercia o la gravedad.

Existe la posibilidad, muy utilizada, de incluir bonus

Lo siguiente a estudiar es el movimiento del juego en sí. Es decir, si tendrá pantalla fija o con movimiento. En el primer caso no hay ningún problema, ya que usando la función *put_screen()* pondremos un fondo de pantalla, que a la vez lo será de nuestro juego. Si se opta por el *scroll*, o movimiento de pantalla, se dispone de la función *start_scroll()*, que cumple este cometido. Se puede optar por mover el *scroll* automáticamente, es decir, que siga al proceso protagonista. Pero también se puede hacer de modo manual, retocando los campos x0, y0, x1 e y1 de la estructura *scroll*. En cualquier caso, la forma de programar esto es sumamente sencilla.

Seguimos hablando de movimientos, en este caso el de los enemigos. Por un lado, podemos hacer que el tipo de movimiento, sea del tipo trayectoria. Para facilitar la programación de estas trayectorias se puede construir un mapeador, que nos permita colocar los enemigos en su posición inicial, y luego indicar el movimiento que realizarán. Este tipo de movimiento puede ir mediante una fórmula, o simplemente mediante coordenadas relativas. Es decir, indicar los puntos por donde pasará el enemigo.

También se puede optar por un tipo de movimiento más *inteligente*. A este tipo lo denominaremos de seguimiento. Consistirá en ir directamente a por la nave protagonista, cogiendo sus coordenadas y haciendo que los enemigos apunten a las mismas y avancen. Se puede poner alguna pared, aunque dificultará la programación del juego, facilitará la labor del jugador. Por último quedan los enemigos que no se mueven, o lo que es lo mismo, inmóviles. Aunque de este tema, se hablará más adelante.

Ahora pasemos a hablar de los disparos. Todos ellos, normalmente, deben tomar las coordenadas del protagonista

El mundo de los videojuegos

Se ha abierto una nueva etapa dentro de estos artículos, con los que os entretenemos cada mes. El denominador común de todos ellos es que tratan videojuegos, comentando sus fundamentos, viendo su problemática y dando solución a los problemas. La lista completa de esta serie de artículos es la que viene a continuación, aunque siempre puede que haya cambios de última hora, ahí va dicha lista:

- Arcade de plataformas.
- Shoot'em'up.
- Puzzles y tetris.
- Comecocos y juegos especiales.
- Juegos de cartas y de mesa.
- Aventuras conversacionales.
- Rol / RPG.
- Juegos de lucha.
- Simuladores deportivos.
- Simuladores de vuelo.
- Simuladores en general.
- Estrategia.
- Juegos tipo DOOM



como posición inicial. Luego moverse por la pantalla según el movimiento estipulado, para al fin comprobar si ha desaparecido de pantalla, etc. Dentro de los disparos podemos hablar primero del más simple, el único, el más típico. Se puede optar por un disparo múltiple, en este caso se debe programar el movimiento de cada bala y luego crearla. Para crear distintas balas, con un sólo proceso y variar su comportamiento, podemos usar el paso de parámetros.
Otro tema, hablando de disparos, es el de los misiles. Es este caso se debe optar por el tipo de misil a usar. Dentro de los tipos posibles nos encontramos primeramente con aquel que se comporta como un disparo normal, pero con mucha más potencia destructiva. Llevado esto hasta los límites, nos encontraríamos con los disparos de megabombas, es decir, aquellos que destruyen gran cantidad de enemigos a la vez. Aunque se podría convertir estas megabombas en misiles.
Por último, nos encontramos con los misiles más difíciles de programar, los conocidos como de seguimiento o misiles inteligentes. En estos casos se debe tener en cuenta que hay que seleccionar un enemigo para que el misil lo siga. Además, hay que tener cuenta que no se pueden enviar dos misiles a un mismo enemigo, por lo que se deberá tener control sobre los misiles enviados.
Además tenemos los disparos de los enemigos, que se pueden comportar de forma parecida a los disparos descritos para la nave protagonista. Aunque lo que normalmente se hace es que los disparos enemigos se dirijan hacia la nave protagonista. Es decir, que a la hora de ser lanzados tomen las coordenadas actuales del protagonista y se dirijan hacia dicho punto. La programación de este tipo de disparo no tiene mucha complicación. Pero si se quiere dificultar realmente la labor del jugador, los disparos que son más difíciles de evitar son aquellos que hacen un seguimiento continuo del protagonista. Es decir, siguen a éste todo el rato, aunque si no se quiere hacer las cosas tan difíciles, en vez de asignar el ángulo donde se va a dirigir directamente se puede usar la función *near_angle()*, lo que hará que se dirija poco a poco, e incluso que se ponga a dar vueltas alrededor de la nave.
Ahora pasemos al tema del escenario, si es que se va a diseñar nuestro juego con esta capacidad. Una vez diseñado el escenario, se puede hacer que éste sea peligroso, o simplemente sea una pared sin ningún peligro. Tanto en un caso como en el otro, se deben usar mapas de durezas para indicar dónde hay pared y dónde no. Como se dijo, un mapa de durezas es un gráfico que tiene pintado de un color los huecos, y de otro las paredes. Únicamente se debe comprobar en qué posición se encuentra el protagonista dentro del mapa de durezas para saber si está o no encima de una pared. Otro método menos fiable es crear el escenario como un proceso y usar la función *collision*.
Y también hablando de escenarios, nos encontramos con los enemigos de escenario. Son estáticos, no se moverán de su posición, su programación es sencilla, tanto si usamos o no el *scroll*. En los dos casos, la única diferencia entre estos enemigos y los otros es que estos no varían sus coordenadas durante la ejecución.
Cambiando de tema, seguiremos hablando de las colisiones y los choques. Como anteriormente se dijo, existen dos maneras de detectar choques con el escenario, bien con los mapas de durezas, bien usando la función *collision()*. Para los enemigos, disparos y demás objetos del juego, la única forma será usar esta función, que detecta el choque entre dos procesos.
Ya por último queda hablar de los bonus. Se consiguen, por ejemplo, al eliminar una horda entera de enemigos. Suele dar más opciones de destrucción, como multidisparos, o vidas, o también algún tipo de escudo. Y hablando de escudos, para programar estos lo que se debe tener, aparte del gráfico que lo simule, es una variable de estado dentro del proceso protagonista que indique si la nave es vulnerable o no. Otro tema a tener en cuenta es que si las naves poseerá energía o simplemente podrá ser destruida en el primer disparo. Para manejar la energía se puede crear una variable local del mismo nombre.
Y para finalizar quedan los enemigos fin de fase y los cambios de fase. En el último caso, únicamente es necesario dos variables para controlar todo. Una que indique el número de fase y otra que indique si se ha acabado o no. Comprobando esta última variable se sabrá si el jugador ha acabado o no. He incluso si lo programamos correctamente, mediante un valor determinado, podemos conocer de qué modo ha acabado. En cuanto a los enemigos de fin de fase, son enemigos especiales que aparecen al final de cada nivel. Normalmente tienen más energía que los otros y se les debe disparar más veces. Sobre su comportamiento, se pueden usar las técnicas descritas para los demás enemigos, y usarlas con este tipo teniendo siempre en cuenta las diferencias entre los mismos.

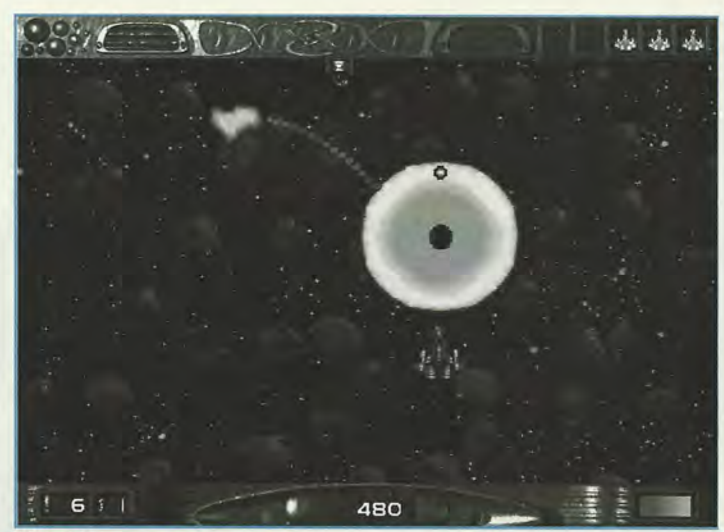


FIGURA 3.

Grandes nombres en el Shoot'em up

Ha habido grandes nombres dentro de los Shoot'em up desde tiempos inmemoriables. Desde los míticos Invaders o Galaxians ha pasado mucho tiempo, y otro de los nombres que destacó dentro de este campo fue el R-Type, juego de Spectrum que hizo historia.
Como protagonista de este tipo de juegos, aunque normalmente han sido marcianos y naves, también se han usado otros elementos. Éste es el caso de juegos como 1942, 1943 y toda la saga, donde se usaban aviones. Como movimiento original de este tipo de juego se podría nombrar el Asteroids. En él la nave giraba y avanzaba.
Además, las capacidades Shoot'em up (que más o menos quiere decir "dispara a todo lo que aparezca", se pueden ver otros juegos que en mayor o menor medida lo usan. Es el caso, por ejemplo, de Jungle Strike, donde se combinan secciones arcade con otras más estratégicas.

Resumiendo

Bueno, ya nos despedimos por hoy, el mes que viene comentaremos otro estilo en cuanto a videojuegos. Si tenéis alguna duda, os podéis poner en contacto con el autor en el E-mail: tizo@100mbps.es, donde dentro de lo posible intentaremos solucionarla. El mes que viene nos vemos por estas páginas, que os DIVirtáis programando.

Autor: Carlos Glez. Morcillo
cgonmor@jet.es

Anarkanoid, el machacaladrillos sin reglas

Como la mejor forma de saber hacer juegos es ponerse a practicar, en esta sección todos los meses incluiremos el código fuente de un juego completo junto con un informe técnico de cómo se hizo e información útil para los desarrolladores.

Este mes abrimos una nueva sección en Game Developer, donde incluiremos códigos fuente completos de varios juegos. Ésta es una de las mejores formas de aprender secretos de diversos programadores; además, viendo el código fuente completo podremos practicar y modificarlo como nos venga en gana. El videojuego de este mes es similar al legendario Arkanoid; consiste en dar con una pala a una pelota para romper los ladrillos de la parte superior de la pantalla. Está desarrollado en Turbo Pascal 7 sin utilización de BGI, y el modo de vídeo utilizado es el 13H (320x200 a 256 colores). Para las rutinas que tienen que ser muy rápidas (como volcado de pantallas virtuales a pantalla real) se ha utilizado Ensamblador, pero únicamente en las partes en que era estrictamente necesario.

Al estar programado en Turbo Pascal, la traducción a otros lenguajes como C o DIV es bastante sencilla, y de esta forma ampliaremos nuestros horizontes. Como en una página es imposible explicar el desarrollo de un videojuego completo, hemos incluido en el directorio de *fuentes* (*/informe*) un documento de 17 páginas (formato Word 97) donde se explica detalladamente cómo se fue desarrollando el videojuego desde el principio, pegas que se plantearon, soluciones, etc. Algunos de los temas tratados en el informe son:

- Descripción y utilización del modo de vídeo 13H.
- La paleta gráfica, inicialización y efectos.
- El problema del retrazado y su solución.
- El ratón, implementación.
- Estructura general del juego.
- Colisiones, el algoritmo de Bresenham.

Cualquier programador, por muy novato que sea, podrá entender fácilmente el código fuente de este juego y profundizar en sus conocimientos. El juego puede mejorarse en varios aspectos, que dejamos como ejercicios propuestos a nuestros lectores:

- Incorporación de efectos de sonido.
- Música ambiental de fondo (por ejemplo, incluirla con Midas).
- Mejorar el algoritmo de colisiones.
- Incluir premios cuando se rompan algunos ladrillos (que la pala aumente o disminuya su tamaño, vidas extras, aumentar o disminuir la velocidad de la bola, en fin, como en el Arkanoid original).
- Mejorar el aspecto gráfico del juego, etc.

Todo lo que se nos ocurra. Dejamos el juego abierto a vuestras propias ideas. Además, podéis contactar con el autor del juego para presentarle vuestras dudas, quejas o lo que queráis escribiendo a *cgonmor@jet.es* Esperamos que disfrutéis con esta nueva sección y estad al loro, que pronto vendrán nuevas sorpresas, y nuevos juegos a los que de este mismo modo destriparemos.





Autor: José Antonio Guerra

El wrapper

Este mes vamos a hacer un pequeño descanso y repasar lo que hemos visto hasta ahora. Nuestro *wrapper* ya es bastante útil, así que vamos a hacer una pequeña referencia para su buen uso. También echaremos un vistazo a los recursos que existen en Internet para el desarrollo en DirectX.

Actualmente, nuestro *wrapper* está dividido en tres módulos: la clase **GDDirectDraw**, la clase **GDDDSurface** y la depuración.
A partir de ahora, nuestro trabajo se centrará en completar la serie con la implementación de los *clippers*, el manejo de paletas, los dispositivos de entrada y los de sonido.

DEPURACION

Una buena parte del trabajo de desarrollo de cualquier programa se invierte en la depuración. Para nuestro *wrapper* hemos creado una macro que nos permite visualizar mensajes en la ventana de depuración de Visual C.

PRINTD(cadena)

Cuando invocamos a PRINTD, éste manda la cadena que le hayamos pasado, junto con información sobre el fichero y la línea de código fuente desde el que se le llamó a la ventana de depuración. Aquí vemos un ejemplo de la salida generada por PRINTD:

****> GDDDSurface::setColorKey(DWORD color) -> D:\Curso DirectX\Ejemplos\Entrega5\GDDDSurface.cpp (360)*

FIGURA 1.PAGINA PRINCIPAL DE DIRECTX EN MICROSOFT.



La salida contiene: primero, la cadena «***>», que nos sirve para diferenciar salidas de depuración nuestras de las de DirectX y así identificarlas mejor; después, la cadena pasada a PRINTD; tras la cadena «->» encontramos el fichero desde el que se llamó a la macro; y finalmente, entre paréntesis, el número de línea. Podéis echar un vistazo a la línea 360 del fichero *GDDDSurface.cpp* para ver cómo se invocó a la macro.
Es recomendable usar PRINTD para saber por dónde está discurriendo nuestro programa. Por eso, en el *wrapper* usamos PRINTD antes de constructores, destructores y funciones complejas. También es recomendable utilizarlo en caso de error leve en el programa.
El buen empleo de estas sencillas técnicas nos evitará muchas horas de depuración, ya que en muchos casos evitaremos tener que ir trazando línea por línea, casi a ciegas. Tenemos también que tener en cuenta que no podemos ir ejecutando línea a línea una aplicación DirectDraw en pantalla completa, con lo que las dudas que pudierais tener respecto a esta técnica hayan sido ya despejadas.

CLASE GDDDIRECTDRAW

Esta clase encapsula un objeto **IDirectDraw2**, facilitando tareas como el establecer modos de vídeo, cooperatividad y *flipping*.
El *wrapper* posee aún ciertas limitaciones. No es capaz todavía de manejar modos gráficos en ventana y los modos creados en pantalla completa deben tener al menos un *back buffer*. Estos inconvenientes se irán resolviendo durante el curso.
Vamos a ver los distintos métodos:

- ***GDDirectDraw():*** Es el constructor por defecto. Simplemente crea el objeto **IDirectDraw2** para su posterior utilización.
- ***GDDirectDraw (HWND hwnd):*** Igual que el constructor. Además, se inicia con un *handle* a la ventana de nuestra aplicación. Este *handle* es necesario para que el *wrapper* sea funcional.
- ***GDDirectDraw (HWND hwnd, int width, int height, int bpp = -1, bool fullScreen = TRUE):*** Este constructor inicia el *wrapper* y establece un modo de vídeo. Ver el método *setMode* para la lista de parámetros.
- ***~GDDirectDraw():*** Destructor del *wrapper*. Se encarga de eliminar todos los objetos creados.
- ***void setHWND(HWND hwnd):*** Esta función sirve para indicar el *handle* de ventana a usar. Suele usarse tras crear el objeto mediante el constructor por defecto, ya que es necesario el *handle* para que el *wrapper* funcione.
- ***HWND getHWND(void):*** Este método sirve para obtener el *handle* asignado a este objeto.

Ficheros del Wrapper

- Clase GDDirectDraw
  - *GDDirectdraw.h*
  - *GDDirectdraw.cpp*
- Clase GDDDSurface
  - *GDDDSurface.h*
  - *GDDDSurface.cpp*
- Herramientas de depuración
  - *GDDebug.h*
  - *GDDebug.cpp*
- Métodos auxiliares
  - *Ddutils2.h*
  - *Ddutils2.cpp*

Listas de correo DirectX

Angelic-coders

Http://www.angelic-coders.com/
DirectXDev@angelic-coders.com

Ésta es una magnífica lista de correo sobre DirectX, en la que participan activamente varios miembros del propio equipo de desarrollo de las librerías. Es bastante visitada, por tanto prepárate a recibir unos 50 mensajes diarios si no activas el modo digest, el cual te permite recibir un correo diario conteniendo todos los mensajes enviados a la lista ese día.



FIGURA 2. EN PLENA SESION DE DEPURACION.

- ***HRESULT getError(void):*** Este método nos va a resultar muy familiar. Devuelve el último error producido dentro del *wrapper.*
- ***bool isFullScreen(void):*** Devuelve *True* si el modo de vídeo es en pantalla completa, y *False* si es en ventana.
- ***HRESULT getDC(HDC* hdc):*** Este método permite obtener el DC del *back buffer*, para poder dibujar en él mediante el GDI.
- ***HRESULT releaseDC (HDC hdc):*** Libera un DC previamente obtenido con *getDC.*
- ***HRESULT flip(void):*** Efectúa el *flipping* entre el *back buffer* y el *front buffer.* El método espera a que pueda realizarse el *flip*, restaurando *surfaces* si éstas se pierden.
- **LPDIRECTDRAW2 get(void):** Sirve para obtener el objeto **IDirectDraw2** que encapsula el *wrapper*, para así poder efectuar tareas no implementadas en esta clase.
- **GDDDSurface* getBack(void):** Esta función devuelve el *back buffer* (nótese que devuelve un objeto **GDDDSurface**).
- **GDDDSurface* getPrim(void):** Este método devuelve el *front buffer.*
- **HRESULT setMode(int width, int height, int bpp = -1, int backbuffers = 1, bool fullScreen = TRUE):** Esta función establece un modo de vídeo, teniendo en cuenta las limitaciones del estado de implementación actual del *wrapper.* Veamos para qué sirve cada parámetro:
  - ***width***: Indica el ancho de modo de vídeo (en pixels).
  - ***height***: Indica el alto del modo de vídeo.
  - ***bpp***: Indica la profundidad de color del modo deseado(expresado en bits por píxel). Si solicitamos un modo en ventana, este parámetro será ignorado.
  - ***Back buffers***: El número de *back buffers* que deseamos que tenga nuestro modo de vídeo. En caso de indicar un modo en ventana, estos *buffers* serán emulados por el *wrapper.*
  - ***fullScreen***: Indica si queremos el modo de vídeo en pantalla completa o en ventana.

CLASE GDDDSURFACE

Esta clase encapsula un objeto **IDirectDrawSurface3**, y permite liberarnos de muchas de las tareas repetitivas que conllevan la gestión de este tipo de objetos, implementando las funciones elementales en el proceso de dibujo.

A este *wrapper* le falta aún la creación de más tipos de constructores, así como un constructor de copia. Aun así, las funciones *create()* suplen la carencia de dichos constructores, por lo cual son perfectamente funcionales.

Todas las funciones *create(...)* borran el contenido del *wrapper* y lo reinician al ser llamadas.

- ***GDDDSurface():*** Es el constructor por defecto. No hace nada salvo inicializar el *wrapper.*
- ***~GDDDSurface():*** El destructor del *wrapper.* Libera todos los objetos necesarios.
- ***HRESULT create(GDDirectDraw* owner, LPDIRECTDRAWSURFACE lpDDSurf):*** Esta función inicia el *wrapper* a partir de un objeto **IDirectDrawSurface.**
  - ***owner***: Puntero al objeto ***GDDirectDraw*** al que pertenece la *surface.*
  - ***lpDDSurf***: Puntero al objeto *surface* a encapsular. Dicho objeto queda inservible a partir de este momento y solamente puede ser usado desde el *wrapper.*
- ***HRESULT create(GDDirectDraw* owner, int numBackBuffers):*** Esta función crea una *surface* con uno o varios *back buffers.*
  - ***owner***: Igual que en el método anterior.
  - ***numBackBuffers***: Número de *back buffers* que deseamos en el *wrapper.*
- ***HRESULT create (GDDirectDraw* owner, const char* fichero):*** Esta función inicia el *wrapper* cargando un fichero BMP en la

IRC-Hispano

Si tenéis alguna duda sobre DirectX y preferís comentarla junto a otras personas en vivo, probad en los siguientes canales:

- *#programacion_stratos:* Este canal está frecuentado por desarrolladores de videojuegos, así que no será difícil encontrar algún programador de DirectX que os resuelva las dudas.
- *#programacion , #informaticos:* Estos canales son de carácter más general, pero os será fácil encontrar a alguien que domine DirectX.
- *#demos:* Aunque la demoscene no esté aún muy metida en desarrollar aplicaciones DirectX, aquí encontrareis verdaderos maestros de la programación gráfica.

surface. La *surface* creada tendrá las dimensiones del BMP, y en caso de pérdida, al restaurarla se volverá a cargar dicho fichero.

- ***owner***: Igual que en métodos anteriores.
- ***fichero***: El nombre del fichero BMP a cargar.

- ***HRESULT createBackBuffer(GDDDSurface *primary):*** Esta función crea un *wrapper* que referencia el *back buffer* de la *surface* dada. Al liberar este *wrapper*, la *surface* encapsulada no se libera.
  - **primary**: *Surface* a partir de cuyo *back buffer* se creará el *wrapper*.
- ***LPDIRECTDRAWSURFACE3 get():*** Esta función permite obtener el objeto **IDirectDrawSurface3** encapsulado, para acceder a él directamente.
- ***GDDDSurface* getBackBuffer():*** Esta función devuelve (si existe) un *wrapper* conteniendo el *back buffer* de este objeto.
- ***HRESULT blitOpaque(int x, int y):*** Esta función dibuja (*blit*) de modo opaco nuestra *surface* en el *back buffer* principal.
  - ***x***: Coordenada horizontal de la *surface* de destino donde se dibujará.
  - ***y***: Coordenada vertical de la *surface* de destino donde se dibujará.
- ***HRESULT blitTrans(int x, int y):*** Esta función dibuja de modo transparente (con el *color key*) nuestra *surface* en el *back buffer* principal.
  - ***x***: Coordenada horizontal de la *surface* de destino donde se dibujará.
  - ***y***: Coordenada vertical de la *surface* de destino donde se dibujará.

Webs de DirectX

- **Página oficial de DirectX en Microsoft**

  Http://www.microsoft.com/directx/default.asp

  En esta web encontraréis las últimas informaciones oficiales sobre DirectX, así como downloads del SDK, run-times y utilidades para las librerías, aparte de mucha información de interés.
- **Enlaces de desarrollo en StRAtOS**

  Http: //www.stratos-ad.com/stratos/enlaces.htm

  En la página de esta asociación podréis encontrar un listado de enlaces a diferentes sitios sobre DirectX, así como a otras páginas relacionadas con el desarrollo de videojuegos.
- **Floating point**

  Http://www.ben2.ucla.edu/~permadi/gamelink/gamelink.html

  Una inmejorable página de enlaces a recursos de programación de videojuegos, entre ellos DirectX.

Cabecera de la clase GDDirectDraw

```
Class GDDirectDraw
{
  public:   //—————————————————
———>> public <<—

            GDDirectDraw();
            GDDirectDraw(HWND hwnd);
            GDDirectDraw(HWND hwnd, int
width, int height, int bpp = -1, bool fullScreen =
TRUE);
            ~GDDirectDraw();

            void setHWND(HWND hwnd) {
_hwnd = hwnd; };
            HWND getHWND(void) const {
return _hwnd; }

            HRESULT  getError(void) const {
return _error; }
            bool isFullScreen(void) const {
return _isFullScreen; }

            HRESULT  getDC(HDC* hdc);
            HRESULT  releaseDC(HDC hdc);

            HRESULT  flip(void);

            LPDIRECTDRAW2 get(void) const {
return _lpDD; }
            GDDDSurface*       getBack(void)
const { return _surfBack; }
            GDDDSurface*       getPrim(void)
const { return _surfPrim; }

            HRESULT setMode(int width, int
height, int bpp = -1, int backbuffers = 1, bool
fullScreen = TRUE);

  protected: //————————————————
>> protected <<—
  private:   //————————————————
->> private <<—

            HRESULT _createDirectDraw();
            HRESULT  _init();
            HRESULT  _createSurfaces(int
backbuffers);

            bool _valid;                    //
Indica si el objeto se ha iniciado
            HWND     _hwnd;
            bool _isFullScreen;
            HRESULT  _error;                //
Ultimo error declarado
            LPDIRECTDRAW2      _lpDD;

            GDDDSurface*
_surfPrim;
            GDDDSurface*
_surfBack;

};
```

FIGURA 3. OTRO PROGRAMA DE EJEMPLO PARA NUESTRO CURSO.





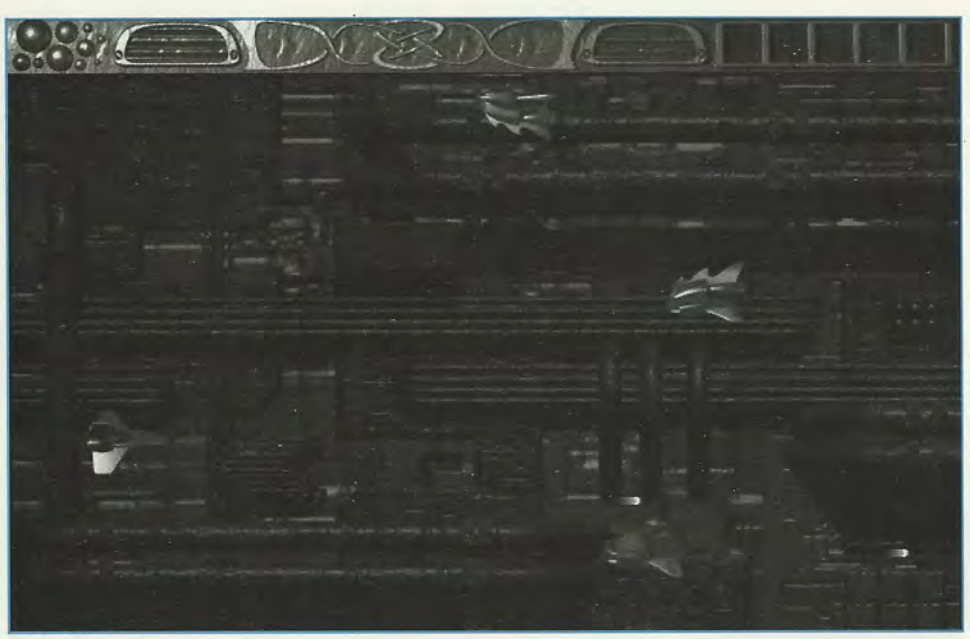


FIGURA 4.

Cabecera de la clase GDSurface class GDDD Surface

```
{
    public:     //——————————————————
————>> public <<—

            GDDDSurface();
            ~GDDDSurface();

            HRESULT create(GDDirectDraw*
owner, LPDIRECTDRAWSURFACE lpDDSurf);
            HRESULT create(GDDirectDraw*
owner, int numBackBuffers);
            HRESULT create(GDDirectDraw*
owner, const char* fichero);

            HRESULT
createBackBuffer(GDDDSurface *primary);

            LPDIRECTDRAWSURFACE3 get()
const { return _surf; }

            GDDDSurface* getBackBuffer();

            HRESULT  blitOpaque(int x, int y);
            HRESULT  blitTrans(int x, int y);
            HRESULT  blitRectTrans(int x, int y,
int x2, int y2);

            HRESULT
setRGBColorKey(COLORREF color);
            HRESULT  setColorKey(DWORD
color);

            HRESULT  getDC(HDC* hdc);
            HRESULT  releaseDC(HDC hdc);

            HRESULT  getError(void) const {
return _error; }

    protected: //——————————————————
>> protected <<—
    private:   //——————————————————
->> private <<—

            HRESULT  reload();
// Recarga la surface con su fichero
            HRESULT  createFromFile();      //
Crea una surface desde un fichero

            HRESULT
_upgradeToDDSurface3(LPDIRECTDRAWSURFAC
E lpDDS);
            void        _release(void);
            void        _init(void);

            GDDirectDraw*       _owner;
            bool _valid;                    //
Indica si el objeto se ha iniciado
            HRESULT  _error;                //
Ultimo error declarado
            char* _fichero;                 //
Nombre del fichero cargado (si hay que

    // restaurar)
            bool _esCopia;                  //
Indica si hay que borrar el objeto

            LPDIRECTDRAWSURFACE3 _surf;
};
```

- ***HRESULT blitRectTrans(int x, int y, int x2, int y2):*** Esta función dibuja de modo transparente nuestra *surface* en el *back buffer* principal, ajustándose a las dimensiones que le pasemos. Sirve por ejemplo para hacer zoom.
  - *x, y*: Esquina superior izquierda del rectángulo donde se dibujará la *surface*.
  - *x2, y2*: Esquina inferior derecha del rectángulo donde se dibujará la *surface*.
- ***HRESULT setRGBColorKey(COLORREF color):*** Esta función establece el *color key* mediante la función *DDSetColorKey2* de **ddutil2.cpp**.
  - *color*: Color RGB que servirá de color transparente. Esta función efectúa transformación de color dependiendo del modo de vídeo utilizado.
- ***HRESULT setColorKey(DWORD color):*** Este método establece el valor del color transparente para la *surface*.
  - *color*: Valor del color transparente. Esta función no efectúa ninguna transformación, con lo cual deberemos ajustar el valor al formato de píxel actual, o bien usarlo para modos de paleta.
- ***HRESULT getDC(HDC* hdc):*** Este método permite obtener el DC de la *surface*, para poder dibujar en ella mediante el GDI.
- ***HRESULT releaseDC(HDC hdc):*** Libera un DC previamente obtenido con *getDC*.
- ***HRESULT getError(void):*** Este método devuelve el último error producido en el *wrapper*.

RECOPILANDO MAS INFORMACION

La mejor fuente de información sobre DirectX es, sin duda, Internet. Hemos hecho un hueco en este artículo para daros algunas pistas sobre la localización de estos valiosos bienes.
Entre las distintas opciones existentes, hemos compilado una pequeña lista de recursos en la que caben páginas web, listas de correo, grupos de noticias e incluso canales de IRC. No os perdáis ninguna de estas referencias si queréis saber todo acerca de la tecnología DirectX. Para dudas, sugerencias o cualquier tipo de críticas os facilito mi dirección electrónica.

Jose Antonio Guerra Pablos
balder@mindless.com

Grupos de News sobre DirectX

Grupo oficial DirectX de Microsoft
Servidor: *msnews.microsoft.com*
Grupo: microsoft.public.win32.programmer.directx
Este foro es una buena elección para estar al día en DirectX. Es bastante frecuentado, así que preparad vuestros módems para recibir todos los mensajes.

Autor: Carlos Gzlez. Morcillo

Diseño y optimización de páginas web

Si la cara es el espejo del alma, entonces la página web es el espejo de nuestro grupo de desarrollo. La velocidad a la que baje y su diseño serán decisivos a la hora de que los usuarios se queden a «bucear» en nuestra información.

Este artículo no pretende dar una iniciación al lenguaje HTML (para eso ya aprendimos con el artículo de Leticia Krahe publicado en Game Developer número 3), sino más bien consejos y ejemplos prácticos para mejorar la página web de nuestro grupo de desarrollo. En muchos grupos se plantea el problema de quién debe ocuparse del mantenimiento y elaboración de la página web. Si bien la mejor opción es que haya una persona dedicada exclusivamente a esta labor, esto nos puede suponer demasiado gasto y en la mayoría de los casos se hace cargo una persona que tiene otras funciones en el grupo. ¿Y quién debe ser? ¿Un programador, un grafista, un músico? Normalmente se suele encomendar al programador adentrarse en esos parajes, pero no es la mejor elección. Una página sobre todo tiene que entrar por los ojos (lo cual no significa que tiene que recargarse de imágenes), y el más adecuado para dar vistosidad a una web es un grafista. Esto no quiere decir que el resto de miembros del grupo no colaboren con el desarrollo de la web. De hecho, el programador podría hacer alguna aplicación en Java o CGI por ejemplo.

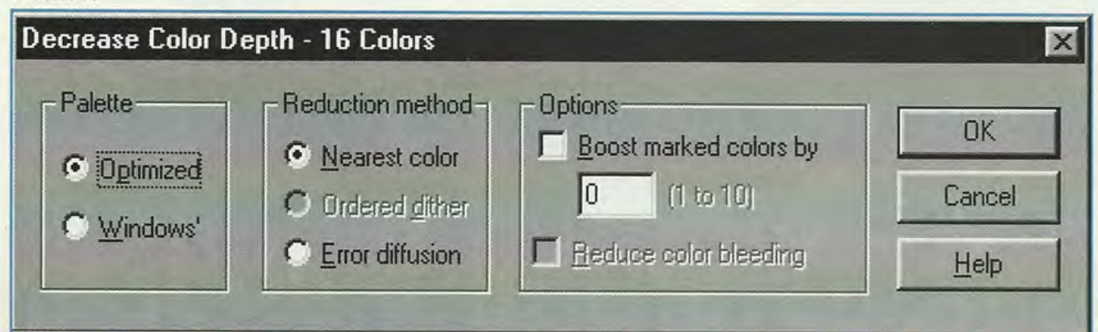
Los modernos editores de páginas web, con la filosofía WYSIWYG (*What You See Is What You Get*, lo que ves es lo que obtienes) facilitan en gran medida la creación de documentos HTML sin tener casi ni idea de la enorme variedad de etiquetas que hay en este protocolo. Para los ejemplos, usaremos el editor que viene incorporado en Netscape Communicator versión 4.02.

INICIOS

Antes de ponernos a escribir en el editor como locos, deberemos pensar sobre qué vamos a incluir en nuestra página y qué diseño va a tener. Sería muy recomendable elaborar un guión entre todos los miembros del grupo y discutir qué información se va a incluir (texto y programas para bajarse), cómo se va a diseñar la página, etc. Es muy molesto visitar una web y no encontrar la información distribuida de alguna forma coherente. Una vez decidido todo esto, debemos tener en cuenta algunas recomendaciones:

- La web debe actualizarse cada cierto tiempo. Para mantener la atención de nuestros visitantes, deberemos aportarles datos nuevos. Una buena costumbre es incluir una página de novedades, en donde detallaremos por orden cronológico las mejoras y actualizaciones que ha sufrido la web. Otra opción muy interesante es la elaboración de un formulario para contactar con los usuarios de nuestra página y conocer sus gustos y opiniones.
- El contenido gráfico tiene una gran importancia. La principal diferencia entre una web profesional y una de aficionado es la incorporación o no de gráficos propios. Pero cuidado, el abuso en la utilización de gráficos no es muy recomendable, ya que la carga de imágenes aumenta el tiempo de "bajada" de una página. El tamaño máximo de una página (entre texto e imágenes) no deberá superar los 60 Kbytes (siendo esta cifra de por sí elevada). Esto se debe a que muchos usuarios de Internet poseen una conexión lenta y un tiempo de espera superior a 15 ó 20 segundos, lo que puede incitarles a abandonar la visita a nuestra página. Así pues, es muy importante optimizar al máximo nuestras imágenes para reducir el tiempo de carga. Si queremos incluir imágenes grandes con una resolución muy buena, es aconsejable utilizar *Thumbnails* (o catálogos en pequeño), y ofrecer las mismas imágenes en pequeño para orientar al visitante sobre qué va a obtener si desea cargar la imagen definitiva.
- Debemos variar el tamaño y el color de la letra para ayudar al navegante en la búsqueda de la información que necesite. La forma de leer una web es distinta a la lectura de un documento impreso. La forma en la que se desplaza una web (en vertical), añadido al condicionante de que la factura telefónica va aumentando, contribuye a que se lea *a saltos* y prácticamente sólo los titulares. De esta forma todas las páginas deben tener algo de valor, y no se debe incluir texto de relleno.
- Tenemos que pensar que la mayoría de los navegantes no tienen ese último Plug-In que

FIGURA 1.



tú has conseguido, y muchos de ellos conexiones lentas. Usar texto alternativo en las imágenes es una buena práctica, y respecto al tema de los Plug-Ins, lo mejor es que aprendamos a usarlos y les demos tiempo a nuestros navegantes a conseguirlos. Si nos decidimos a utilizarlo, incluyamos siempre un enlace a una página donde conseguirlo.

- No hay nada más frustrante que seguir un enlace a una página "en construcción" es preferible que no funcione el enlace hasta que la página no esté lista, más todavía si tienes que cargar el típico gráfico del obrero trabajando con la pala.
- Evitar construir páginas muy largas, además de ser incómodas para el visitante, las páginas cortas facilitan las tareas de mantenimiento y actualización.

PRACTICANDO UN POCO

Como ejemplo de este artículo haremos un sencillo formulario para nuestra página web. Sin duda alguna no es la mejor forma de elaborar un formulario, pero es la más sencilla y el resultado es bastante efectivo. Como hemos indicado antes, las imágenes deberán ocupar el menor espacio posible, para que bajen rápido. Deberemos elegir entre los dos formatos que soportan la inmensa mayoría de *browsers*: el GIF y el JPG.

GIF VS. JPG

El formato GIF tiene un sistema de compresión LZW bastante sencillo, que no degrada la imagen original, pero tampoco permite comprimir mucho ésta. Tendremos la limitación de trabajar con 256 colores, pero podremos realizar animaciones y fijar un color de transparencia. Por el contrario, el formato JPG utiliza un sistema de compresión más sofisticado, nos permite utilizar 24 bits de color y las compresiones son mucho mejores que en GIF. Como defectos del JPG podemos indicar que la imagen va perdiendo calidad según se aumenta la compresión, y además no podemos fijar colores transparentes ni hacer animaciones.

Llegados a este punto, nos preguntamos ¿cuándo utilizar cada formato? Pues utilizaremos GIF siempre y cuando las imágenes no sean excesivamente grandes y el paso a 256 colores (o menos) se pueda hacer sin perder calidad. Siempre que utilicemos una imagen de fondo de página, tendremos que usar transparencias, por lo que el formato GIF es el más adecuado. Para imágenes grandes, con muchos colores y gran definición, utilizaremos JPG, ya que su compresión es notablemente superior.

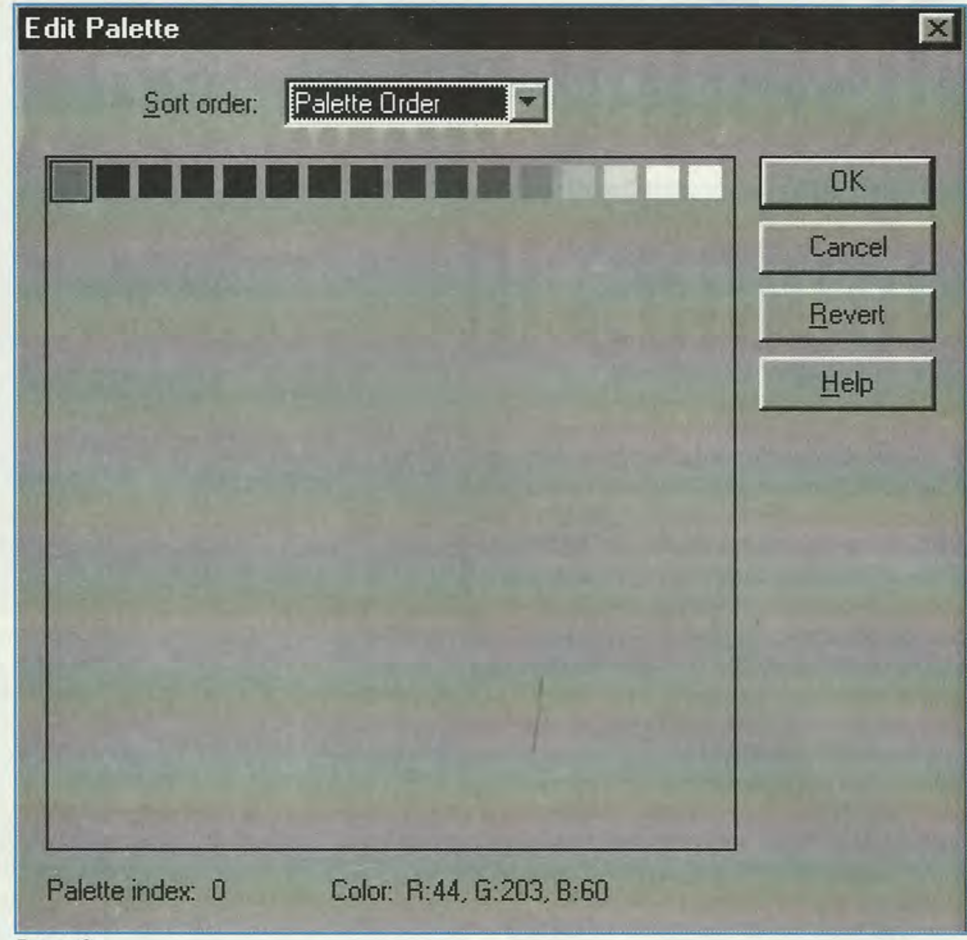


FIGURA 2.

ALGUNOS CONSEJOS PARA OPTIMIZAR IMAGENES

Para imágenes GIF, deberemos ajustar el número de colores al mínimo, ya que influye en el tamaño final de la imagen. También podemos cambiar la escala de la imagen inicial por otra más pequeña (tamaño en pixels) para ahorrar bytes, y en el editor agrandar la imagen al tamaño que deseemos. De esta forma perderemos algo de definición, pero ganaremos en tiempo de bajada de la página. En GIF animados no deberemos realizar esta operación de cambio de escala, ya que algunos *browsers* dan errores. Al hacer la imagen original más pequeña debemos tener en cuenta que esta operación debe realizarse antes de disminuir el número de colores. Es decir, primero cambiaremos la escala de la imagen a 24 bits de color y luego procederemos a disminuir el número de colores. Aunque en un principio pueda parecer que quedaría igual, el orden de los factores sí altera el producto, ya que en *true color* se realiza perfectamente el cambio de escala y en *indexado* no.

Otro aspecto a tener en cuenta es el color de fondo de la imagen antes de hacerlo transparente. Si la imagen de fondo de nuestra página es de tonos negros o grises (como la del ejemplo), deberemos construir nuestras imágenes sobre un fondo de tono gris oscuro.

FIGURA 3.



FIGURA 4.



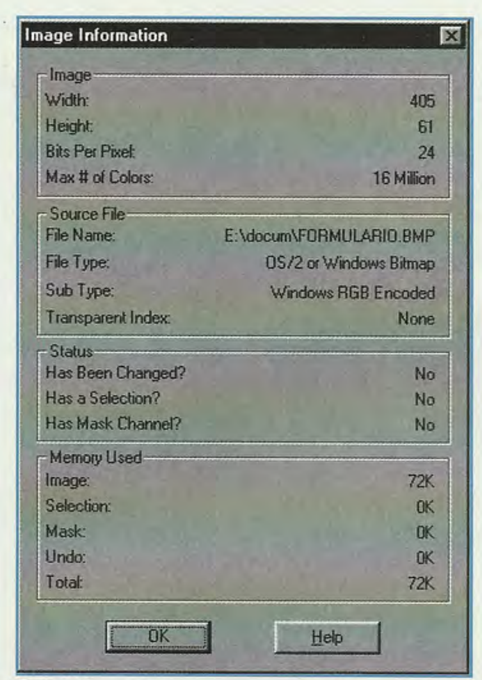


FIGURA 5.

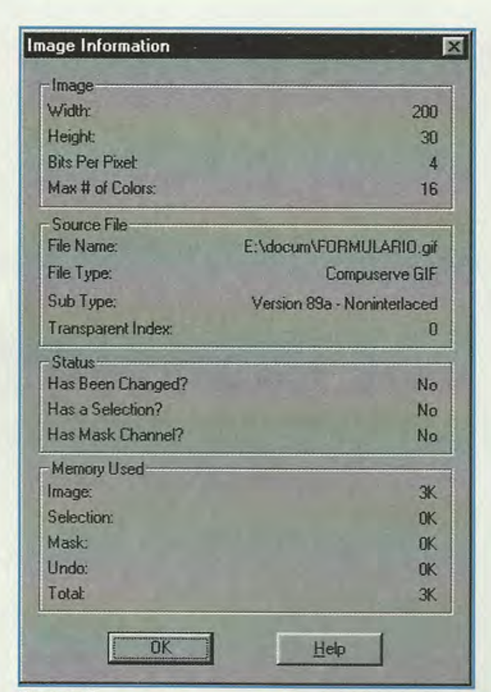


FIGURA 6.

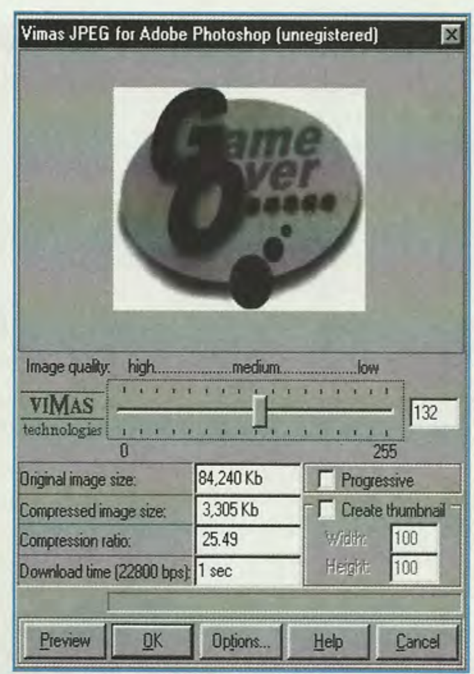


FIGURA 7.

Así, cuando fijemos un color como transparente, no se fugarán pixels de un color no deseado.

En el ejemplo de la página, hemos reducido el tamaño original de una imagen en formato semi-crudo (BMP) de 72 Kbytes a 3 Kbytes. Para ello hemos utilizado el programa Paint Shop Pro que se incluye en el CD que acompaña a la revista. Cargamos la imagen original, y cambiamos la escala a 200x30 pixels, ahorrando así la mitad de espacio en disco. Para ello iremos al menú *Image-resample* y ahí indicaremos la dimensión a la que queremos reducir la imagen. Si tenemos seleccionada la opción *Maintain Aspect Ratio* tan sólo tendremos que modificar una de las dos dimensiones y la imagen no se deformará. Una vez realizado el cambio de escala, disminuiremos el número de colores y fijaremos un color como transparente. Para disminuir el número de colores, iremos a *Colors-Decrease Color Depth* y reducimos a 16 colores. ¿A cuántos colores debemos reducir la imagen? Pues al mínimo posible sin que la imagen pierda demasiada calidad. Todo es cuestión del diseñador. Aunque la imagen pierda un poco (no demasiado) deberemos sacrificar un poco de calidad y reducir así el tamaño de la imagen, los visitantes de nuestra web lo agradecerán. Tenemos la imagen a 16 colores, ahora fijaremos un color como transparente. Iremos a *Colors-Edit Palette*, elegiremos el color de fondo de la imagen original (que era negro) y lo cambiaremos por un color chillón. Así tendremos la imagen con un fondo que destaque. Nos quedamos con el número de color que ocupa dentro de la paleta (en la misma ventana, abajo *Palette index*). Ya tenemos la imagen lista para ser salvarla. Elegimos el formato GIF y pinchamos sobre *Options-Set the transparence value to palette entry*, el número que hay que poner a continuación es el del color que hemos fijado antes como transparente, el número que ocupaba dentro de la paleta. En nuestro caso era el 0. Hecho esto, guardamos la imagen.

Para incluir animaciones en nuestras páginas, como hemos indicado antes, debemos utilizar el formato GIF. En el mercado hay una extensa gama de programas para construir GIFs animados. Uno de los más sencillos de utilizar y muy potente es el *Gif Movie Gear*, que incluimos en el CD de la revista. Insertaremos los distintos fotogramas que componen una animación uno por uno. Para ello, iremos al menú *File/Insert Frame* y seleccionaremos las diferentes imágenes que compondrán la animación. Si no están colocados en el orden que deseas, bastará con que pinches sobre el *frame* descolocado y lo arrastres a la posición que necesitas. Una vez organizados todos los *frames*, indicaremos el tiempo de espera que habrá detrás de cada uno de ellos. Bastará con que pinchemos (para seleccionar) sobre cada *frame*, y pongamos el tiempo en el recuadro que hay al lado del reloj del menú superior. Una vez ajustada la velocidad optimizaremos la animación. Reduciremos paletas al máximo (siempre que no pierda demasiada calidad) y optimizaremos la animación (el programa lo hace automáticamente). Para ello iremos a

FIGURA 8.



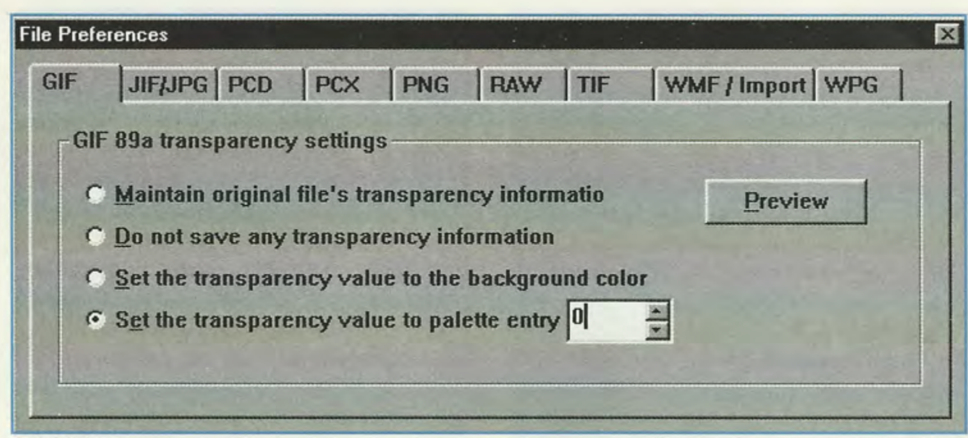


FIGURA 9.

Tools / Reduce Palettes y a *Tools / Optimize Animation* respectivamente. La verdad es que no hay mucho que explicar sobre este programa, porque es bastante sencillo de utilizar y muy intuitivo. En una tarde, seguro que aprenderéis a sacarle el máximo partido.

HAGAMOS UN EJEMPLO EN NETSCAPE

Un formulario es muy útil en todas las webs. Así, los visitantes de nuestras páginas nos informarán de lo que más les gusta de la página o las partes que se pueden mejorar y, en general, sus gustos y preferencias. La elaboración de un formulario puede ser complicada, pero aquí os damos una forma sencilla y efectiva de elaborar vuestros propios formularios. Pondremos un ejemplo desde Netscape. Iremos a *File/Edit Page* y comenzaremos con el diseño de nuestra página. Las zonas donde el visitante escribirá o realizará operaciones con ellas serán los *HTML tags*. No creemos que sea necesario explicar cómo se ha hecho todo el formulario. Lo mejor es que lo abráis (viene en el CD de la revista) y experimentéis con las distintas posibilidades. De todas formas, haremos un pequeño resumen:

Para que nuestros visitantes incluyan datos como nombre, número de teléfono, profesión, etc., disponemos de un tipo de entrada denominada «*text*»:

<INPUT TYPE=text NAME="Nombre" SIZE="50" MAXLENGTH=100>

En *<NAME>* pondremos un identificador, *<SIZE>* será el tamaño del "cuadradito en blanco" que aparecerá en pantalla y con*<MAXLENGTH>* especificaremos la longitud máxima de texto que se podrá introducir.

Para opciones de elección sencilla (tipo Sí o No), disponemos de «*checkbox*»

<INPUT TYPE=checkbox NAME="DesarrolloYFuentes" VALUE="SI">

En *<VALUE>* pondremos el texto que recibiremos en caso de que se active la caja. Otro tipo de entrada de datos es la persiana desplegable con varias opciones cuyo esquema general es el siguiente:

<SELECT name=Opinión>
<OPTION>
<OPTION>
....
<OPTION>
</SELECT>

Y por último el área de texto libre, donde el usuario podrá escribir cualquier comentario, duda o sugerencia respecto de nuestra web.

<TEXTAREA NAME="Comentario" COLS=60 ROWS=10 WRAP="PHYSICAL">

Como hemos comentado antes, la mejor forma de aprender es practicar con estos comandos y familiarizarnos con ellos. Por cierto, si pensáis utilizar el esquema de nuestro formulario, ¡no olvidéis cambiar el E-mail al que será enviado el formulario en el primer tag!

Nada más por este mes. Nos vemos muy pronto con un curso de animación de personajes, técnicas y ejemplos. Hasta entonces, un saludo y a machacar pixels!

FIGURA 10.

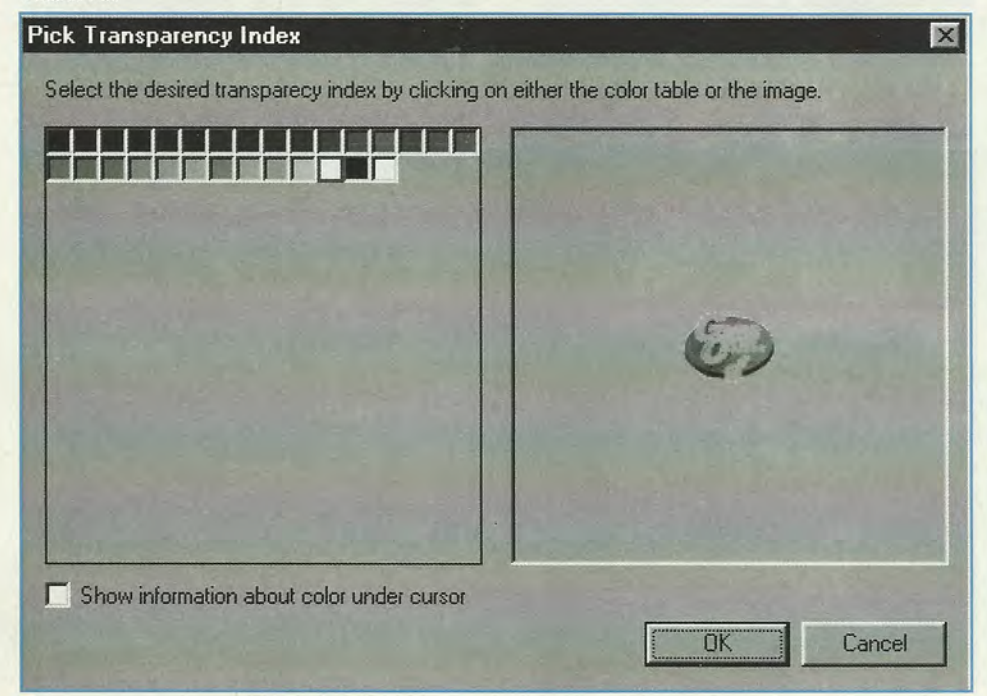


FIGURA 11.

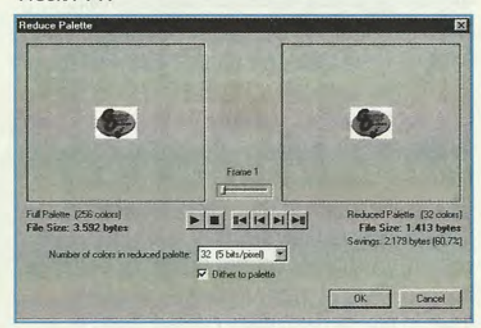


FIGURA 12.

